TimeKeeper: A Lightweight Virtual Time System for Linux

Jereme Lamps, David M. Nicol, Matthew Caesar
University of Illinois at Urbana-Champaign
{lamps1, dmnicol, caesar}@illinois.edu

ABSTRACT

We present TimeKeeper: a simple lightweight approach to embedding Linux containers (LXC) in virtual time. Each container can be directed to progress in virtual time either more rapidly or more slowly than the physical wall clock time. As a result, interactions between an LXC and physical devices can be artificially scaled, e.g., to make a network appear to be ten times faster with respect to the software within the LXC than it actually is. Our approach also supports synchronized (in virtual time) emulation, by grouping LXCs together into an *experiment* where the virtual times of containers are kept synchronized, even when they advance at different speeds. This has direct application to the integration of emulation and simulation within a common framework.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; D.4.4 [**Operating Systems**]: Communications Management—*message sending, communication management*; D.4.8 [**Operating Systems**]: Performance—*measurements, simulation*; I.6.3 [**Simulation and Modeling**]: Applications—*Miscellaneous*

Keywords

Simulation, Emulation, LXCs, Virtualization, Time Dilation, CORE, Linux Kernel

1. INTRODUCTION

Virtual machine managers (VMM) multiplex the execution of virtual machines (VM), i.e., software stacks, in such a way that the VMs behave as though they are running on individual pieces of hardware. A question of great interest to us is how the advancement of time in a VM is perceived. For example, suppose that an application in one VM sends a message to another, and includes in that message the time at which the message was sent. What is the value of that time variable? In most systems the system clock will give the time, but the system clock in a VMM *typically* reflects time advancement of the VMM, not its VMs. A Xen VM has an associated *domain time*, which reflects the amount of wall-clock time that the VM has received. The domain time advances with the system clock while the VM is served, and stops advancing when it no longer has CPU service. However, domain time is used in Xen for scheduling, and not as a measure of virtual time.

The idea for embedding Xen in virtual time was originally expressed in the context of testing distributed applications [7]. The basic idea is to make virtual time in a VM advance more slowly than real time, in order to make the (real) network connected to the VM appear to be performing faster. The approach associates with each VM an integer-valued *time dialation factor*, or TDF. A TDF of n reduces the advancement rate of a VM in time by a factor of n; for example, a TDF of 2 makes virtual time in the VM advance at half the rate of wall-clock time. This approach (and subsequent ones [6]) rescale a VM's notion of time with reference to a *physical* network, in order to emulate a seemingly accelerated rate of interaction between the VM and the network.

We are motivated by a different objective, to virtualize time in a VM in order integrate its behavior with a network *simulator* such as S3F [11] or ns-3 [13]. This goal raises new considerations. A network simulator can represent a much larger infrastructure than a real network in a typical lab, which creates the need to emulate in virtual time many VMs; this in turn raises the importance of minimizing overhead. In particular, we want to have VMs "jump" over epochs in virtual time where nothing of interest occurs, rather than rescale time and have the VM crawl through the epoch just to advance its clock to its next interaction with the network simulator. An example is a web-server, whose behavior is to wait for a request, formulate and issue a database query, wait for the IO system's response, report the response and then wait for another request. Depending on what the experiment is measuring, the VM might be directed to reset its virtual time to the time of a request or IO completion and completely bypass epochs where the process is suspended. Another new consideration is the need to advance a group of VMs through virtual time so that their virtual clocks are closely synchronized, even if they advance those clocks at different rates by virtue of different TDFs.

A final goal is to bring virtual time to the Linux kernel in a minimally invasive way, that exposes an API to support our motivating problem of integrating emulation and simulation, and is general enough to support other uses of virtual

SIGSIM-PADS'14, May 18–21, 2014, Denver, CO, USA.
Copyright 2014 ACM 978-1-4503-2794-7/14/05 ...$15.00.
http://dx.doi.org/10.1145/2601381.2601395 .

time. These considerations have led us to develop *TimeKeeper*, a small set of modifications to the Linux kernel that allows for the creation of LXCs, each with their own virtual clock. There are two main distinctions separating this project from previous ones. First, our approach becomes an integral (and very small) part of the Linux kernel, giving it the potential to become mainstream. Second, our approach is much lighter weight than Xen [2] or OpenVZ [16] (admittedly at the cost of less generality than Xen). Finally, our approach to virtual time synchronization is more sophisticated than that which has been applied to Xen's to date, and has greater flexibility of interaction between emulation and simulation than previous solutions based on OpenVZ.

2. RELATED WORK

Related work exists in the area of simulation/emulation, and in the area of virtual time. We discuss these separately.

2.1 Simulation/Emulation

In simulation computer systems are modeled entirely in software. Simulation has the attractive property of being scalable and repeatable. There are various simulation tools available today, such as J-Sim [9], ns-2 [12], OMNeT++ [14], and ns-3 [13]. J-Sim is a component-based simulation environment, where each link, node, and protocol is a component. Components have ports associated with them, and a component contract describes how data should be handled if it arrives at a specific port. Event executions are in real-time, thus improving the fidelity of the simulation. OMNet++ and ns-2 are both popular discrete event simulators. Both simulators are written in C++, while ns-2 provides the interface through OTcl. In ns-2, models are flat, meaning that creating subnetworks is not possible. On the other hand, OMNeT++ supports a hierarchical module structure which makes it easier to develop complex models in a methodical manner. Many papers have compared the performance of ns-2 and OMNeT++, concluding that ns-2 is not nearly as scalable or easy to use as OMNeT++ [10, 15, 20]. Also, there is ns-3, a discrete event simulator that is aimed to overcome ns-2's shortcomings. It is developed in C++, and it is designed to be modular, scalabale, and extensible. Papers have conducted studies testing the performance of ns-3 and other simulators, concluding ns-3 to be the most efficient [20]. In addition, ns-3 is a hybrid approach, allowing for emulation as well.

In contrast to simulation, emulation involves a testbed or a physical network to provide more realistic results. Two common testbeds that provide emulation are Emulab [21] and PlanetLab [17]. Emulab provides the experimenter with the ability to create arbitrary networks, and allocates specific nodes from the testbed for a specific amount of time. The experimenter can run specific operating systems on the hardware, and is granted *root* access. This allows for controllable and predictable experiments; however, it is limited by the size of the testbed, and may not be suitable for all types of tests. On the other hand, PlanetLab is a global research network consisting of nodes throughout the globe. At the time of writing, PlanetLab consists of 1181 nodes at 572 sites. A distinction between PlanetLab and Emulab is that PlanetLab gives you an LXC on various nodes, while Emulab will give you sole access to the machine. Therefore, experiments on PlanetLab will not be reproducible, because other users may be running experiments on the same nodes simultaneously.

There also exist hybrid solutions, supporting both simulation and emulation, such as the Common Open Research Emulator (CORE) [1] and ns-3 [13]. This allows the simulator to be interfaced with real-world communication systems for more realistic measurements.

2.2 Virtual Time Systems

There have been many recent papers dealing with giving systems a sense of virtual time, e.g., [3, 5, 6, 19, 22]. DieCast [6] makes modifications to the Xen hypervisor to give VM's a concept of virtual time. DieCast also scales the performance of physical hardware components. This is a useful option if you want to create an experiment where the number of nodes in the experiment is greater than the number of nodes in your testbed. SVEET! [3] is a performance evaluation testbed running on Xen-based VMs that implements time virtualization techniques if the simulation is overloaded. It sets a static TDF to slow down both the simulator and the VMs. Our work differs from both DieCast and Sveet, as we use lightweight LXCs with Linux kernel modifications instead of Xen-based VMs. We can also dynamically change TDFs, as well as support the synchronization of LXCs virtual times, even if they have different TDFs. In some ways our approach resembles that of Zheng et al., [22] who developed a virtual time system for simulation and emulation using OpenVZ. Like our solution, they modified time-related system calls to return virtual time as opposed to the system time. However, our work is different, as it uses LXCs, and brings the notion of a time dialation factor to the forefront. The OpenVZ system scales measured elapsed time as we do, but that scaling factor is fixed.

3. DESIGN

We designed TimeKeeper with three objectives. First, we wanted to develop a lightweight solution. This minimizes overhead for time-dilated processes. Next, we wanted a simple solution which would allow researchers to create and test their own time-dilated processes. Finally, we wanted TimeKeeper to easily integrate with existing emulators/simulators. We next expand on these objectives.

3.1 Lightweight

We want to spin up many time-dilated processes simulataneously, with minimal overhead. An attractive option is to use Linux Containers (LXCs) [8], a virtualization method which allows multiple individual Linux instances to be running on a single host while sharing the kernel. LXC produces less overhead than traditional virtual machine monitors, such as Xen [2] or VMWare [18], as they require separate kernels for each VM. We also attempted to minimize the number of changes made to the Linux kernel. For example, to support basic time-dilation within a process, we need only add 36 bytes to the process' *task_struct* (the *task_struct* is a data structure in Linux that stores information about a particular process). These changes required modifying only 7 files in the kernel and adding fewer than than 100 lines of code. To support advanced time-dilation features, such as running processes with different TDFs within the same *experiment*, we developed a linux kernel module which may be dynamically loaded into the kernel at runtime.

3.2 User Interface

In order to make TimeKeeper easy to use, we developed a simple and intuitive application programming interface (API) to create and manage time-dilated processes. The presented API is simply a subset of functions which TimeKeeper provides. The API exports the following functions:

- **clone_time(unsigned long flags, float dilation, int should_start):** causes a new process to be cloned from the calling process. You can set specific flags just as you would in the *clone()* system call. The *dilation* argument is the dilation factor of the new process, and the *should_start* argument will start the new process immediately with a value of 0, and not start the new process with a value of 1. This is useful if you wish to clone numerous processes, and then start them all at the same time (as in an *experiment*).
- **start_experiment(int count, ...):** causes a series of cloned processes all to be started at the same time. *Count* represents the number of processes in the experiment, followed by a variable number of pid integers.
- **dilate(int pid, float dilation):** changes the dilation factor of a process. *Pid* represents the unique ID of the process, and *dilation* is the new dilation factor of the process. This can be called on both processes that were created through the *clone_time()* function, as well as general processes.
- **freeze(int pid):** stops the process from executing. The time at which it stopped executing is remembered.
- **unfreeze(int pid):** allows a previously frozen process to continue executing. In between the time in which the process was frozen and unfrozen, the process does not perceive the passage of time. For example, if a process was frozen at time *t=10 seconds*, and unfrozen at time *t=20 seconds*, the process will resume at time *t=10 seconds*.
- **leap(int pid):** changes the container's virtual time to be identical to that of the container with id *pid*. Applied to a frozen process, it causes that container to leap over an epoch of virtual time, without modification to its TDF.

3.3 Ease of Integration

Finally, we wanted to be sure TimeKeeper could be integrated with other simulation or emulation systems. As proof of concept, we integrated TimeKeeper with CORE [1]. We chose to initially integrate with CORE as it already uses LXCs. Therefore, changes needed to the framework would be minimal. In addition, the framework itself is highly customizable. With only a few modifications to the graphical user interface (GUI) to allow setting dilation factors, plus minor changes to the backend we were able to run simple time dilated experiments. The results can be found in section 5.

4. IMPLEMENTATION

We sought a solution that can run different LXCs at different TDF's. We wanted to be able to run LXCs individually with no synchronization, or grouped together in an experiment where processes with different TDF's must progress uniformly together in virtual time. The following subsections describe modifications to the linux kernel and the development of a linux kernel module respectively that will provide needed functionality.

Algorithm 1: Finding current time of a time dilated task

```
def gettimeofday(tv):
    Data: struct timeval which gets returned to the user
    if task→v_s_t > 0:
        new_p_p_t = now() - task→v_s_t
        new_p_v_t = (new_p_p_t - task→p_p_t)/d_f +
        task→p_v_t
        time = new_p_v_t + task→v_s_t
        task→p_p_t = new_p_p_t
        task→p_v_t = new_p_v_t
        tv = ns_to_timeval(time)
    else:
        Do normal gettimeofday(tv)
```

Figure 1: Pseudocode For Gettimeofday Algorithm

4.1 Kernel Modifications

We added only 36 bytes (5 variables) to the linux *task_struct* in order to give each dilated process its own perception of time. The variables added are:

- 4 bytes *dilation_factor (d_f)* represents the time dilation factor of the process.
- 8 bytes *virtual_start_time (v_s_t)* represents the point in virtual time (in ns) at which a process starts progressing by its TDF.
- 8 bytes *past_virtual_time (p_v_t)* represents how much virtual time has passed since the last time the process inquired about the current time.
- 8 bytes *past_physical_time (p_p_t)* represents how much physical time has passed since the last time the process inquired about the current time.
- 8 bytes *freeze_time (f_t)* is used to determine if a process is currently frozen or not. A value of 0 means it is not frozen, where a value greater than 0 represents the point in time (in ns) in which a process was frozen. This is a variable internal to TimeKeeper.

The *gettimeofday()* system call was modified to return the virtual time for a process if it has a *virtual_start_time* set; if the *virtual_start_time* is not set, then *gettimeofday()* performs normally. The pseudocode for the gettimeofday modifications can be found in Figure 4.1.

Consider a quick example for clarification, using a process with a TDF of 2. Note this means for every 2 seconds of clock time, the process will perceive only 1 second of virtual time. We assume the process is started at the system time of 20 seconds. At this point in time, *d_f=2, v_s_t=20, p_v_t=0, and p_p_t=0.* Suppose this process performs a computation for 10 seconds, and then calls *gettimeofday().* Following the pseudocode, a new_p_p_t will be calculated by subtracting the current system time from the v_s_t. So the new_p_p_t = 30s - 20s = 10s. A new_p_v_t is then calculated by finding the time which has elapsed since the last past_physical_time, scaling it appropriately based on the TDF, and finally adding it to the last past_virtual_time.

Thus, new_p_v_t = (new_p_p_t - p_p_t)/d_f + p_v_t = (10s - 0s)/2 + 0 = 5s. So the virtual_time = v_s_t + new_p_v_t = 25 seconds, which is the correct virtual time for the described scenario. Note, before *gettimeofday()* returns, new_p_p_t and new_p_v_t are stored into p_p_t and p_v_t respectively. At the end of this function, the state of the process is: *d_f=2, v_s_t=20s, p_p_t=10s, and p_v_t=5s* and the global time is 30s. Now assume the process runs for an additional 20 seconds, and checks its time once again. new_p_p_t = 50s - 20s = 30s and new_p_v_t = (new_p_p_t - p_p_t)/d_f + p_v_t = (30s - 10s)/2 + 5 = 15s. So the virtual_time returned is 20s+15s = 35s. As you can see, this is consistent with what is expected, as the process was started at 20 seconds, and has been running with a TDF of 2 for 30 seconds of physical time.

In order to accurately maintain the process' perception of time, we can not simply alter the *gettimeofday()* system call, we must modify system calls such as *sleep()* and *poll()* as well. The *sleep()* system call takes an integer as an argument, which represents the number of seconds the program should sleep before it continues its execution. We modified the *sleep()* system call such that it is scaled with the calling process' TDF. For example, if a process with a TDF of 2 calls *sleep(10)*, it will sleep for 20 seconds of wall clock time. However, due to its TDF, it will believe it slept for 10 seconds. The *poll()* system call waits for a set of file descriptors to become ready so it may perform I/O. *Poll()* takes a *timeout* value as an argument, which corresponds to the minimum number of milliseconds the system call will block. Similarly to *sleep()*, *poll()* was modified so a process with a specified TDF will run as anticipated.

4.2 Kernel Module

Some of the more complicated time dilation functionality was developed in the form of a loadable linux kernel module, e.g., the ability to *freeze* and *unfreeze* a process' advancement in virtual time. In addition, TimeKeeper is able to synchronize containers, so we are able to group processes together with different TDF's and still manage to insure their virtual times are synchronized.

To *freeze* or *unfreeze* a process, TimeKeeper makes use of a variable that was added to each process' *task_struct*: *freeze_time (f_t)*. If the user wishes to *freeze* a process, its *f_t* is set to the current, non-dilated system time, and a SIGSTOP signal is sent to the process, removing it from the CPU and putting it in a stopped state. When the user wishes to *unfreeze* a frozen process, the process' *p_p_t* is updated to reflect the amount of physical time the process was frozen *(p_p_t = p_p_t + (current_system_time - f_t))*. A SIGCONT signal is then sent to the process, allowing it to run on the CPU once again. Finally, *f_t* is reset to 0. To continue the example in the previous section. Assume the process was frozen immediately after it last checked its time (virtual_time=35s, system_time=50s). The current state of the process is: *d_f=2, v_s_t=20s, p_p_t=30s, p_v_t=15s, f_t=50s.* The process is first frozen for 10 seconds, then unfrozen and immediately checks the time. When it is unfrozen, the *p_p_t* is changed to *(p_p_t + (current_system_time - f_t)) = (30s + (60s-50s)) = 40s.* When it checks the time with the updated *p_p_t* value, it returns 35s, therefore not recognizing any time has passed since it was frozen.

In addition to freezing and unfreezing a process' perception of time, TimeKeeper is also responsible for grouping processes with different TDF's into a single *experiment*, where all of the processes virtual times progress uniformly.

TimeKeeper maintains a linked list of all processes in the experiment, a tunable knob called a *timeslice* which specifies the amount of phyiscal time the *leader* LXC should be allowed to run in each interval, and another tunable knob that specifies how many processors can be used for LXCs in the experiment (DED_CPU). We would set DED_CPU to be two CPUs less than the total number of CPUs in the system. This would allow standard background tasks to still run successfully, even when performing a CPU-intensive experiment. When an *experiment* is initialized, TimeKeeper determines the process with the highest TDF, known as the *leader*. Knowing the *leader* is a necessity, as the *leader's* virtual time will be progressing slower than any other process in the experiment. Therefore, we need to scale down the running time of other processes in the experiment accordingly. For example, if the *leader* has a TDF of 2 and there is another process with TDF of 1, the process with a TDF of 1 will need to run for one half the time the *leader* runs.

Once the *leader* has been determined, each process is dedicated to a specific CPU, where multiple processes may be dedicated to the same CPU, and set to have a scheduling policy of SCHED_FIFO (first-in first-out). We set each process' scheduling policy as SCHED_FIFO so it will have priority over other tasks not in the experiment, as well as not get pre-empted until we say so. Each process will receive a fraction of the *timeslice* in which it will be allowed to run on its dedicated CPU, this fraction is based on the process' TDF in respect to the *leader's* TDF, and maintained by a high-resolution timer (*hrtimer*) [4]. To run a process, it is unfrozen with TimeKeeper's previously mentioned *unfreeze* capability, and its *hrtimer* is set to expire when its fraction of the *timeslice* is up. When the *hrtimer* for a process expires, that process is frozen, and the next process whose turn it is to run on the CPU gets unfrozen and has its *hrtimer* set. When all processes in the experiment have been allowed to run for their fraction of the *timeslice*, the round is up. At the end of each round, the *leader* will be recalculated if new processes were added to the experiment, or if the past *leader* finished executing. Each process' virtual time is compared to the expected virtual time. If a process' virtual time exceeds the expected virtual time, that process will be forced to run for less time in the following round (by setting the *hrtimer* to expire earlier). If a process' virtual time is below the expected virtual time, that process will be allowed to run for additional time in the following round. The next round begins when all processes know how long they should be allowed to run for in the next round. See Figure 4.2 for the basic psuedocode.

5. EVALUATION

In this section, we will discuss our preliminary results regarding the accuracy of *hrtimers*, and our virtual time systems ability to keep the LXCs synchronized. In addition, we look into scalability of the system, the overhead TimeKeeper may create, as well as how efficiently TimeKeeper can keep LXCs running in real-time. Unless otherwise specified, experiments were conducted on a Dell Studio XPS Desktop, with 24 GB of RAM, and 8 Intel Core i-7 CPU X 980's @ 3.33GHz. The machine is running 32-bit Ubuntu with a modified 3.10.9 Linux kernel.

Algorithm 2: Simplified Process Synchronization and Hrtimer Interrupt

```
def synchronize(timeslice):
    expected_time = calcExpectedVirtualTime()
    foreach task in experiment do
        dilated_time = calcDilatedTime(task)
        difference = expected_time - dilated_time
        task→offset = calcOffsetNeeded(task, difference)
    foreach CPU in DED_CPU do
        task = getNextTask(CPU)
        unfreeze(task)
        task→setHrTimer(timeslice - task→offset)
    return
def hrtimerInterrupt(task):
    freeze(task)
    nextTask = getNextTask(task→CPU)
    if nextTask == NULL:
        synchronize(timeslice)
    else:
        unfreeze(nextTask)
        nextTask→setHrTimer(timeslice - nextTask→offset)
    return
```

Figure 2: Pseudocode for LXC Synchronization Algorithm

timeslice	μ	σ
300ms	862ns	1130ns
30ms	401ns	680ns
3ms	341ns	592ns
300μs	523ns	2306ns
30μs	351ns	2128 ns
3μs	481ns	3312ns
1μs	2404ns	4213ns
300ns	2925ns	6012ns

Table 1: Mean and Standard Deviation of Timer Error for Different Timeslice Lengths

5.1 hrtimer accuracy

The effectiveness of TimeKeeper's ability to keep virtual clocks synchronized is highly dependent on the *hrtimers* ability to fire interrupts at precise moments in time. If we want a particular LXC to run for 3μs at a time, then we would want the *hrtimer* associated with that particular LXC to trigger an interrupt as close to 3μs as possible. For the initial test, we set different *hrtimers* to periodically fire at different time intervals (*timeslice*), and measured what time the *hrtimer* interrupt actually fired. We collected 200 data points for every different time interval. From there, we calculated the mean (μ) and standard deviation (σ) of the error. Table 1 presents the results.

Taking the first row as an example, when the timer was scheduled to fire an interrupt every 300ms, on average the interrupt occurred 862ns from what was expected. This is excellent accuracy, there are five orders of magnitude between the error and the *timeslice*. The magnitude of the variation in error is roughly constant; the error size relative to *timeslice*is still an order of magnitude smaller with a 30 micro-second *timeslice*, and is roughly equal with a 3 micro-second *timeslice*. These comparisons tell us something very important about the level of granularity we can effectively use in combined emulation/simulation scenarios. If 10% error in timing is acceptable and a simulated message takes on the order of 100 micro-seconds to pass on the network from one device to another, we can expect to get a little over three *timeslices* in during the message's passage through the network simulator. *This* means that if a container is sensitive to IO from the simulator only at *timeslice* boundaries (as is the case with the virtual-time OpenVZ system), there may be as much as a 33% error in the virtual time at which the container "sees" the message. The take-away message here is that Linux timers are very accurate, but if we are to be able to take advantage of that accuracy when interfacing emulated LXC containers and a network simulator we will have to find a way to integrate simulator time and container time at a finer granularity than the *timeslice*. This constitutes one of our areas of future work.

# of LXCs	timeslice	μ	σ
10	.3ms	596ns	1084ns
10	3ms	685ns	1129ns
10	30ms	1028ns	1766ns
10	300ms	812ns	1447ns
80	.3ms	196ns	375ns
80	3ms	193ns	374ns
80	30ms	258ns	535ns
80	300ms	333ns	628ns

Table 2: Mean and Standard Deviation of Error as a Function of Timeslice and #LXCs

5.2 Synchronization

To integrate our emulation with network simulation we will need to keep LXCs closely synchronized. We performed a set of experiments to evaluate how tightly we are able to do so. In these experiments, TimeKeeper aimed to have each LXC achieve a target virtual time by the end of each *timeslice*. For each LXC and each *timeslice* we measure the deviation of the virtual time the LXC actually achieved at that *timeslice* from the target goal. For each set of experiments we compute the mean error μ and the the standard deviation of the error σ, taken over all LXCs and synchronizations, and observe the behavior of these errors as a function of the number of LXCs and the size of the *timeslice*. Our first round of experiments used the same TDF for all containers; each container was engaged in the compute-intensive task of computing the factorial of a large number.

For the first experiment, we used a TDF of 10 for each container, and recorded measurements for 150 *timeslice* intervals. The results are summarized in Table 2, and reveal some interesting information. First, it demonstrates that TimeKeeper is effective at keeping virtual times synchronized on the *timeslice* sizes used. TimeKeeper is seemingly more effective at keeping the experiment synchronized when the *timeslice* length is 3ms rather than 300ms. At the time of this writing we are unsure of the underlying cause for this difference, and are working at additional instrumentation in an effort to uncover an understandable explanation.

To give better insight into the distribution of error, we also plotted two cumulative distribution functions (CDFs). Figure 3 shows us a CDF when the number of LXCs in the experiment range from 10-80, and the *timeslice* interval is constant at 3ms. Regardless of whether the experiment had 10 LXCs or 80 LXCs, TimeKeeper was able to keep every LXCs virtual time within 4μs of the expected virtual time for more than 90% of each *timeslice* interval. However, this comes at a cost. The more LXCs you add to the experi-

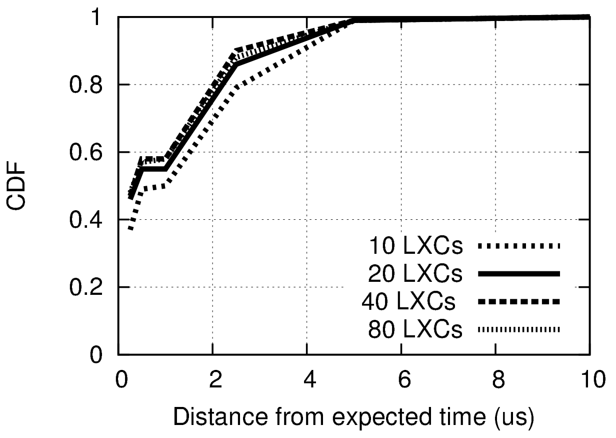


Figure 3: CDF with timeslice=3ms as a function of #LXCs

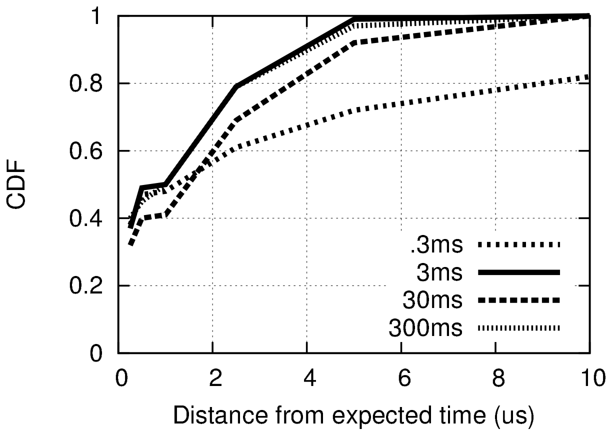


Figure 4: CDF with 10 LXCs as a function of timeslice length

ment, the longer it takes for the experiment virtual time to progress. This will be explored more fully in Section 5.3. Figure 4 shows us a CDF when we have an experiment size of 10 LXCs (where 5 LXCs have a TDF of 10, and 5 LXCs have a TDF of 1), and we vary the *timeslice* interval lengths. In general, TimeKeeper is able to keep the experiment virtual time in sync, but we noticed when the *timeslice* interval is $.3ms$ that it did not perform as well. These results correspond with what we found in Table 1 (where the *hrtimers* were not as accurate at a granularity of $.3ms$ as opposed to higher granularities).

5.2.1 Scalability

Figure 5 demonstrates scalability, plotting how the mean and standard deviation of the error behaves as the number of containers grows. Again we see the interesting phenomena that the error decreases with increasing numbers of containers; the error is also contained almost always to be less than half a micro-second.

We obtained access to a larger machine, with 32 cores and 64Gb of memory. This allowed us to bring TimeKeeper up and observe how many containers we can sustain. We successfully did one experiment using 45,000 containers, which

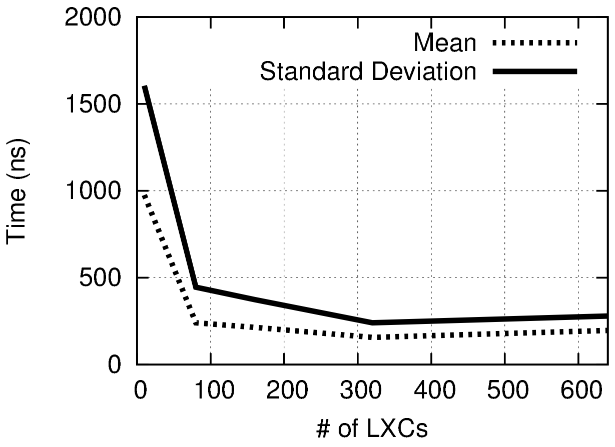


Figure 5: Testing Scalability with a Timeslice of 3ms and a TDF of 1/10

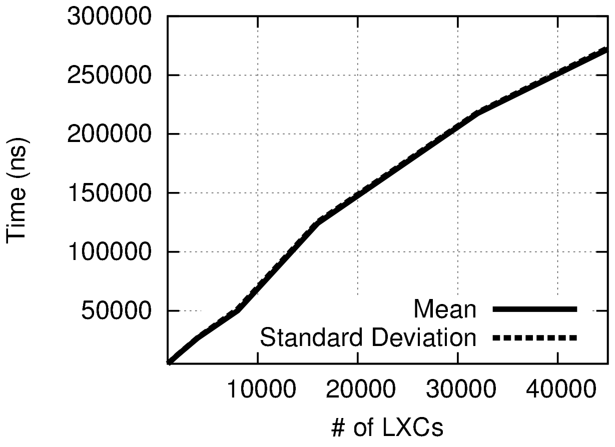
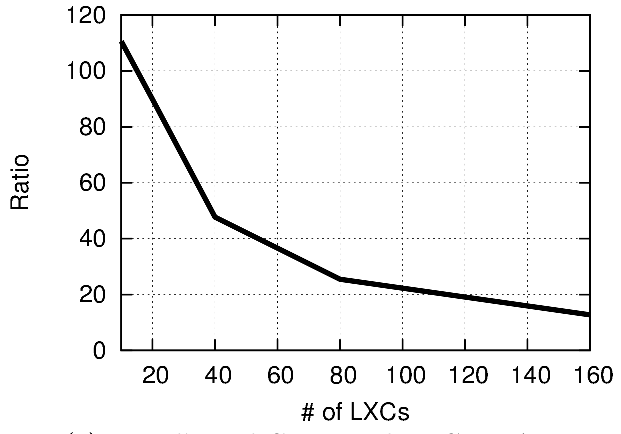


Figure 6: Testing Scalability with the Product of #LXCs and TDF Constant

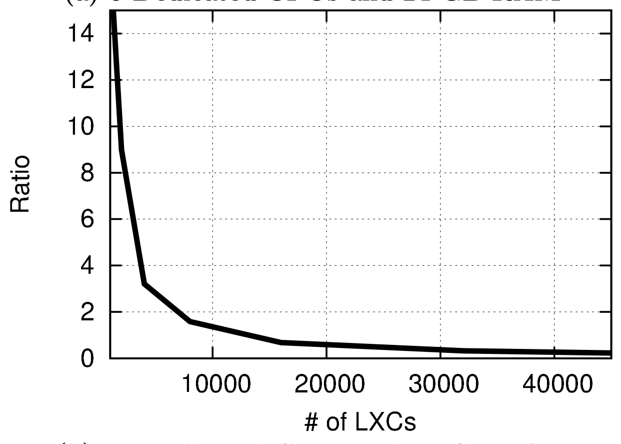
represents two orders magnitude increase of what could be done on that same machine with openVZ containers.

We performed an experiment aimed at measuring the mean and standard deviation of the time error found when TimeKeeper tries to keep all LXC containers in an experiment synchronized. For this we keep the product of number of containers with the TDF constant, at approximately 20. The intuition is we are trying to keep the rate (in wallclock time) at which virtual time advances in the system as a whole constant—increasing the number of containers means the number of times a container is given service per unit wallclock time decreases, so each time it gets service it has to advance simulation time farther. Now in these experiments the *timeslice* length is kept constant.

Figure 6 displays the results, and reveals an interesting consequence of the scaling we employ. As the number of LXCs increases, the TDF decreases, which means that the advance in virtual time per unit wall-clock tick increases. The error of timers *in wall-clock time* is unaffected by the number of containers, however this fixed error is *amplified* by the amplification of virtual time advancement. This explains the linear increase in error. We'd get essentially the same curve—but with different y-axis values—by using a different

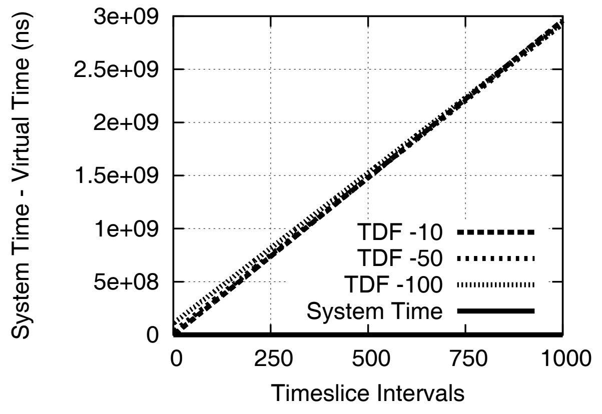


(a) 6 Dedicated CPUs and 24 GB RAM

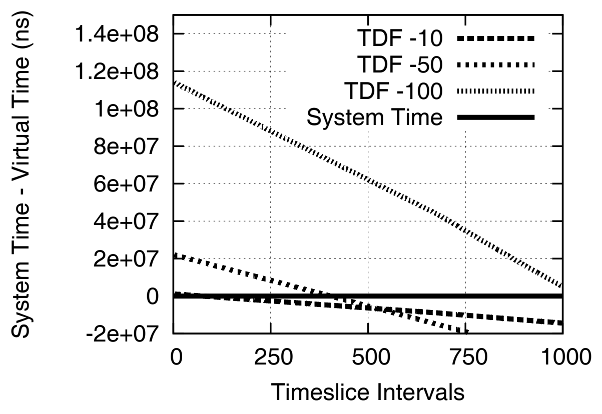


(b) 28 Dedicated CPUs and 64 GB RAM

Figure 7: Overhead Ratio with Timeslice=3ms as a Function of #LXCs



(a) 6/(TDF+1) #LXCs



(b) 6/(TDF+1) +1 #LXCs

Figure 8: Determining Maximum #LXCs Where Real-Time is Maintained

constant product of TDF and #LXCs. A product that is larger by a factor of 10 will yield errors that are a factor of 10 smaller. Two main points should be appreciated from this data. One, that TimeKeeper has managed as many as 45,000 synchronized containers on a commodity server, and second, that the error of timers in real-time has more impact on the errors in virtual time the faster the containers are accelerated through virtual time.

5.3 Overhead

We measured the scheduling overhead of TimeKeeper, by dividing the amount of physical time progression of the leader LXC by the amount of time spent in the synchronization method of TimeKeeper. We call this the *overhead ratio* (OR). The larger the OR value, the more efficient the emulation. We ran multiple experiments with different TDFs and *timeslice* lengths. We learned that as *timeslice* length increases, so does the OR. This is intuitive, as a larger *timeslice* will call TimeKeeper's synchronization function less frequently.

Figure 7(a) shows how the OR changes as the number of LXCs in an *experiment* increases. For this particular experiment, the *timeslice* was set to 3ms, and we scaled the number of LXCs from 10-160. As the number of LXCs grew, the OR decreased. This is because TimeKeeper must manage more LXCs, and managing these additional LXCs results in more overhead. This overhead can be reduced by dedicating more CPUs to the LXCs in the experiment.

The overhead ratio calculated on a machine with 32 cores (28 dedicated cores) and 45,000 LXCs was **.23** and is shown in Figure 7(b). This is to be expected, and reducing that overhead is a topic of future study. For example, the task of setting the timers for all LXCs to initiate the *timeslice* can be accomplished with a tree structure, rather than the serial structure our current implementation employs.

5.4 Maintaining Real-Time

We also wanted to determine how efficient TimeKeeper is at keeping LXCs running in real-time. When we say real-time, we mean that for every instant in time, all LXCs in the experiment will have a virtual time that is greater than or equal to the system time. Obviously, we will only be able to keep an experiment in real-time if all of its TDFs are all less than or equal to 1. For the experiment, we assumed all LXCs have the same TDF. Therefore, the maximum number of LXCs in an experiment we can keep in real-time is: N/TDF, where N is the number of dedicated CPUs on the machine, and we are assuming no overhead. However, our system does have overhead, so our experiment will determine just how close we can get to this upper bound. We ran experiments with 6 dedicated CPUs, a *timeslice* of $3ms$, and TDFs of 1/10, 1/50, and 1/100 with increasing numbers of LXCs per experiment, until we found the tipping point (the point where we could no longer keep the experiment as a whole in real-time). We calculated the virtual time of each LXC and compared it to the system time at the end of each *timeslice* interval. Our results are in Figure 8. We found the maximum number of LXCs to be: $6/(TDF + 1)$, any more LXCs cause a tipping point and the experiment can no longer be kept in real-time. Figure 8(a) displays the virtual time of the experiment with respect to the system time using this tipping point. As you can see, all experiments virtual time is *increasing* linearly in respect with the system time. Figure 8(b) displays the same thing, but this time, adding just 1 more LXC to each experiment, ie: $6/(TDF + 1) + 1$. This is obviously the tipping point, as all three experiments virtual time is now *decreasing* with respect to the system time.

6. CONCLUSION

We introduced TimeKeeper: a lightweight, simplistic, and easily integrated solution to provide LXCs with their own view of virtual time. TimeKeeper was integrated into the CORE framework, and collected promising results. We demonstrated TimeKeeper's ability to keep the LXCs virtual time synchronized to within half a micro-second of error. In addition, scalability was also tested, with experiments pushing upwards of 45,000 LXCs, each with their own virtual time, all on a single commodity server.

7. ACKNOWLEDGMENTS

This work was supported in part by the Boeing Corporation, and in part by by the Department of Energy under Award Number DE-OE0000097. [1]

8. REFERENCES

[1] J. Ahrenholz, C. Danilov, R. Henderson, and H. Kim. Core: a real-time network emulator. In *Proceedings of the 2008 International conference for military communications (MILCOM'08)*, 2002.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, 2003.

[3] M. Erazo, Y. Li, and J. Liu. Sveet! a scalable virtualized evaluation environment for tcp. In *Testbeds and Research Infrastructures for the Development of Networks & Communities and Workshops*, 2009.

[4] T. Gleixner and D. NieHaus. Hrtimers and beyond: Transforming the linux time subsystems. In *Proceedings of the Linux Symposium (Ottawa, Ontario, June 2006)*, 2002.

[5] A. Grau, S. Maier, K. Herrmann, and K. Rothermel. Time jails: A hybrid approach to scalable network emulation. In *Proceedings of PADS'08*, 2008.

[6] D. Gupta, K. Vishwanath, and A. Vahdat. Diecast: Testing distributed systems with an accurate scale model. *NSDI'08*, 2008.

[7] D. Gupta, K. Yocum, M. McNett, A. Snoeren, A. Vahdat, and G. Voelker. To infinity and beyond: Time-warped network emulation. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005.

[8] M. Helsley. Lxc: Linux container tools. *IBM developerWorks Technical Library*, 2009.

[9] J-sim official. https://sites.google.com/site/jsimofficial/, 2014.

[10] M. Koksal. A survey of network simulators supporting wireless networks, 2008.

[11] D. Nicol, D. Jin, and Y. Zheng. S3f: The scalable simulation framework revisited. In *Proceedings of the 2011 Winter Simulation Conference*, 2011.

[12] The network simulator - ns-2. http://www.isi.edu/nsnam/ns/, 2014.

[13] The ns-3 project. https://www.nsnam.org/, 2014.

[14] Omnet++ community site. http://www.omnetpp.org/, 2014.

[15] Omnet++ vs ns-2: A comparison. http://ctieware.eng.monash.edu.au/OMNeTppComparison, 2014.

[16] Openvz: a container-based virtualization for linux. http://openvz.org/MainPage, 2014.

[17] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[18] Vmware virtualization software. http://www.vmware.com/, 2014.

[19] E. Weingartner, F. Schmidt, H. Vom Lehn, T. Heer, and K. Wehrle. Slicetime: a platform for scalable and accurate network emulation. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, 2011.

[20] E. Weingartner, H. vom Lehn, and K. Wehrle. A performance comparison of recent network simulators. In *Proceedings of the IEEE International Conference on Communications*, 2009.

[21] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[22] Y. Zheng, D. Nicol, D. Jin, and N. Tanaka. A virtual time system for virtualization-based network emulation and simulation. In *Journal of Simulation*, 2011.

[1]Disclaimer: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.